SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **10**, 353–380 (1998)

Research

Model-based Design of Reverse Engineering Tools

STAN JARZABEK* and GUOSHENG WANG

Department of Computer Science, School of Computing, National University of Singapore, Lower Kent Ridge, Singapore 119260

SUMMARY

Tools built in an *ad hoc* way and without proper models often display problems for both tool users and designers. Firstly, without systematic analysis and good understanding of the underlying software process model, we have little chance to design a tool that will adequately address users' needs. Next, because one tool is often used in many different situations and by people who have different working habits, tools should be flexible and allow a user to customize tool functionalities. *Ad hoc* built tools usually are not flexible enough, as possible variations in tool functions have not been incorporated into the tool architecture to make future customizations possible. Finally, *ad hoc* design practice does not lead to accumulating the tool design know-how, making it difficult to repeat successful solutions and slowing down the process of understanding and improving tool design methods. We applied conceptual modelling in design of tools for software maintenance to alleviate some of the above problems.

In this paper, we describe a model-based method for designing reverse engineering tools. The design starts by modelling low-level source program design models, higher-level design models to be recovered, and heuristic rules a reverse engineering tool uses to recover higher-level designs from lower-level designs. On one hand, conceptual models lead to better understanding of tool requirements. On the other hand, a model-based approach leads to the design of a generic design abstractor, a component of a reverse engineering tool that evaluates reverse engineering heuristics. A generic design abstractor adds flexibility to reverse engineering tools in two ways: (1) we can customize the generic design abstractor to meet the requirements of a reverse engineering project in hand, and (2) a programmer (an end-user of a reverse engineering tool) can define new reverse engineering heuristics and tune-in recovered designs.

KEY WORDS: reverse engineering; conceptual modelling; generic tools

1. INTRODUCTION

Conceptual modelling is an integral part of problem solving and, in general, of any activity that attempts to achieve a goal in a systematic way. In different domains, we

* Correspondence to: Stan Jarzabek, Department of Computer Science, School of Computing, National University of Singapore, Lower Kent Ridge, Singapore 119260. Email: stan@comp.nus.edu.sg

Contract/grant sponsor: National University of Singapore; Contract/grant number RP950616

CCC 1040–550X/98/050353–28$17.50

Received 12 March 1997
Revised 30 April 1998

construct different models depending on the structure of a domain and the intended role of a model (Ohsuga, 1995). The premise of this paper is that conceptual modelling can play a vital role in the design of programming tools. Designing tools involves analysis of a software process to be supported by a tool, behaviour of people who use a tool, information processed and produced by a tool, and other aspects of tool functionality. If we can express the results of this analysis in the form of precise models, we can study and better understand tool requirements. With the help of models, we can start viewing tools in a given category as different variants of the same generic tool concept. This realization may in turn lead us to identifying a generic architecture for a family of tools, based on which we can develop individual tools faster and cheaper. In the past, we saw how this process of growing understanding, modelling and formalisation led to elegant, cost-effective and generic solutions in domains of compilers and database applications.

We have used a model-based approach to build static program analysis, reverse engineering and business re-engineering tools (Jarzabek, 1998; Jarzabek and Tan, 1995; Jarzabek and Ling, 1996). Our tools are flexible from the user perspective and generic from the tool designer perspective. A *flexible reverse engineering tool* allows a programmer to filter and tune in recovered design abstractions. A *generic tool*, on the other hand, allows a tool designer to customize a tool to the specific requirements of a reverse engineering task in hand. A practical reverse engineering tool should be both flexible and generic so that it can evolve in an operating environment. Canfora, Cimitile and de Carlini (1992) identified lack of flexibility and genericity as one of the main reasons why current reverse engineering tools have not been widely adopted in industrial projects.

In this paper, we present a detailed study of model-based design of reverse engineering tools. We have found many papers describing reverse engineering methods and functions of related tools, but only one (Canfora, Cimitile and de Carlini, 1992) addressing the issues of how we can design flexible and generic reverse engineering tools in a systematic way. This paper attempts to fill this gap.

The paper is organized as follows. We start by providing reverse engineering background and reviewing related work. In Section 3, we present the essentials of the model-based approach to designing reverse engineering tools. In Section 4, we explain conceptual notations for modelling program design and, in Section 5, a notation for specifying reverse engineering heuristics. In Section 6, the reader will find examples of how we used our models to formalize object recovery heuristics. in Section 7, we describe an architecture of our reverse engineering tool and in Section 8—the design of a generic design abstractor. Comparison of our tool with other similar systems and concluding remarks end the paper.

2. BACKGROUND AND RELATED WORK

The objective of reverse engineering (Chikofsky and Cross, 1990) is to extract design information, functional specifications and, eventually, requirements from the program code. Reverse engineering techniques can help in maintenance of poorly documented programs and in software re-engineering projects. If, for example, we re-engineer a COBOL application that uses flat files into a relational database application, we would typically start by building an entity-relational data model (Chen, 1976). Analysis and reverse engineering of existing file definitions can help in building such a model.

Experimental studies on program understanding (Brooks, 1983; Estadale and Zuylen, 1993) show that program comprehension involves creating multiple mental models of the software, identifying objects within each model, and establishing how the objects interact within and across models. If those program models are not properly documented, programmers recover them (usually in an incomplete and approximate form) from code and from other sources. Reverse engineering practice is, therefore, as old as programming. As a research discipline, reverse engineering is fairly new and the debate about reverse engineering objectives, potentials and levels of possible automation still continues (Brown, 1993).

Recovering functional abstractions and object-like modules from programs has received much attention. Practical situations in which this technique can be useful include re-documenting programs in an object-orientated way, re-engineering procedural programs into object-orientated architecture (Jackobson and Lindstrom, 1991) and re-engineering existing code for reuse (Canfora, Cimitile and Munro, 1994). Reverse engineering of data is of practical importance as many companies face migration from old databases (or data stored in flat files) to relational or object-orientated databases. Typically, the first step in such migration is recovering of the entity-relationship data model from the existing program code (Arora and Davis, 1985; Darlison and Sabanis, 1993; Yang and Bennett, 1995). Reverse engineering into formal specifications is another promising approach to recovering functional and structural descriptions from the program code (Ward, Calliss and Munro, 1989; Lano, Breuer and Haughton, 1993). In the RECAST method (Edwards and Munro, 1995), COBOL applications are reverse engineered into logical descriptions in the format of SSADM. The Rigi system (Wong *et al*., 1995) concentrates on recovering the architectural structure of large legacy software and provides a user with a flexible interface to customize multiple views of software architecture produced by the system. Research on reverse engineering using AI techniques concentrates on identifying higher-level design abstractions and domain concepts in programs. In DESIRE (Biggerstaff, 1989), the domain model is built first and then code patterns are used to find instances of domain concepts in code. In Recogniser (Rich and Wills, 1990) so-called plans represent commonly used program structures. '*Programming plans* are units of programming knowledge connecting abstract concepts and their implementations' (Hartman, 1992). In the process of program recognition, a program is searched for instances of plans.

Success stories with reverse engineering have been reported (Sneed, 1995; Byrne, 1991), many tools have been implemented both in industry (Rock-Evans and Hales, 1990; Burson, Kotik and Markosian, 1990; Kozaczynski, Ning and Engberts, 1992) and academia (Biggerstaff, 1989; Ward, Calliss and Munro, 1989; Rich and Wills, 1990; Canfora, Cimitile and de Carlini, 1992; Canfora, Cimitile, and Munro, 1994; Lano, Breuer and Haughton, 1993; Edwards and Munro, 1995; Wong *et al*., 1995; Jarzabek and Tan, 1995), but more experimentation is needed to fully explore the potential and limits of automated reverse engineering. Canfora, Cimitile and de Carlini (1992) identified a number of problems that hinder wide adoption of reverse engineering tools. In particular, the authors note that current reverse engineering tools:

(1) fail to identify the right level of details in recovered design and lack proper design presentation strategies,

(2) recover only a small portion of the design information software maintainers need, and
(3) are not flexible enough, and cannot be easily tailored and enriched in the operating environment.

The authors further point out that reverse engineering tools recover design views that are commonly used during forward engineering, such as structure charts. However, the design information that is essential during software maintenance (and, therefore, should be produced by reverse engineering tools) is different from the design information used during forward engineering.

3. AN OVERVIEW OF THE MODEL-BASED APPROACH

In our work, we investigated the model-based approach to the design of reverse engineering tools in order to address some of the problems identified by Canfora, Cimitile and de Carlini (1992). The main objective of our model-based approach is to facilitate design of reverse engineering tools that are flexible and generic, tools that can evolve in the operating environment and grow as our understanding of the reverse engineering discipline advances.

There are many variations in software maintenance and re-engineering projects that affect requirements for reverse engineering tools. Depending on the reason why we do reverse engineering and how we are going to use reverse engineered information, we may need to extract many different types of design views, such as formal specifications (Ward, Calliss and Munro, 1989; Lano, Breuer and Haughton, 1993), structure charts, data flows diagrams, inter-modular data flows (Canfora, Cimitile and de Carlini, 1992), control flow graphs, entity-relationship diagrams and slices of those views. Reverse engineering tools work with different sources (such as database schema, source codes or JCL) and use different reverse engineering heuristics to recover abstract program views. By *reverse engineering heuristics* (a notion similar to candidature criterion (Canfora, Cimitile and Munro, 1994; Canfora *et al*., 1994)) we mean rules of how higher-level design views are to be produced from the low-level ones. Often, we need to export reverse engineered design views to other tools (e.g., into a CASE repository) for further processing and presentation. Addressing each and every one of these possibilities requires a specialized reverse engineering tool (Figure 1).

In model-based design, we analyse and classify these variations so that we can view the design of different reverse engineering tools as an instance of a generic design scenario, based on a generic tool architecture. A generic architecture implements functions common to all the reverse engineering tools, in a parametrized form. We implement a reverse engineering tool by tuning the parameters based on the characteristics of a reverse engineering project in which the tool is to be used.

Of course, not all the components of reverse engineering tools can be generic in the above sense. For example, a front-end that parses sources and extracts low-level program design information from code (e.g., abstract syntax trees, control flow graph, data flow relations, etc.) must be built for each new source language. (The task of building front-ends may be simplified by using parser generators but, for the purpose of our discussion, we exclude external tools such as parser generators from the generic architecture.) The

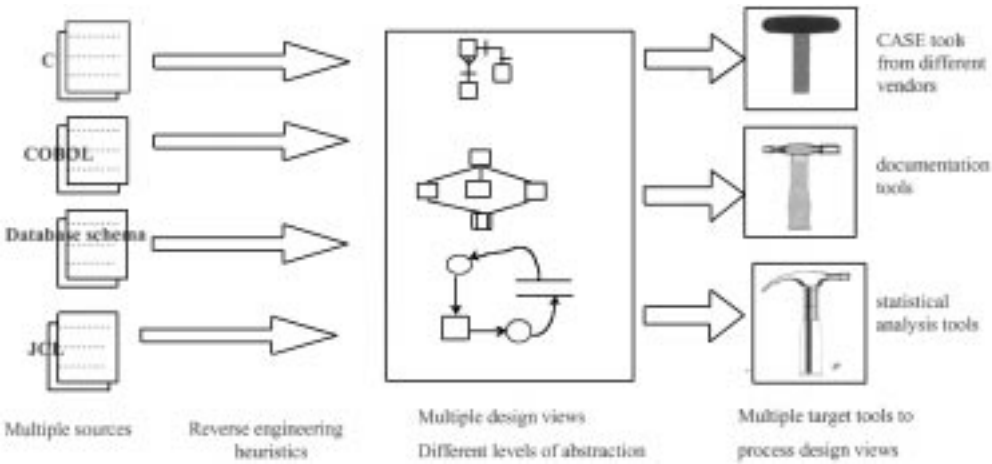


Figure 1. Variant elements in reverse engineering projects and tools

essential component of a reverse engineering tool is a *design abstractor*. The design abstractor is responsible for applying reverse engineering heuristics to derive higher-level design views from lower-level ones. A *generic design abstractor* is one that can be customized to different program design models and different reverse engineering heuristics. It should also be independent of the physical storage for program designs.

With our model-based approach we are addressing the following aspects of tool design:

(1) describing characteristics of software maintenance and re-engineering projects;
(2) describing capabilities of reverse engineering tools in terms of:
    (a) models of low-level program design from which a tool extracts higher-level design views,
    (b) models of higher-level program design,
    (c) reverse engineering heuristics, i.e., rules used to map low-level design patterns into higher-level design views;
(3) designing of a reverse engineering tool architecture and a generic design abstractor. The generic design abstractor should be independent of the major variable characteristics of reverse engineering tasks and tools. These include low and high-level program design models, reverse engineering heuristics and physical storage of low and high-level program design; and
(4) providing a user interface to allow human experts to interact with a reverse engineering tool by analysing and manipulating recovered design models and modifying reverse engineering heuristics.

3.1. A reverse engineering process model

A reverse engineering process model clarifies the context into which reverse engineering tools should fit. Reverse engineering requires heuristic rules to map program structures and low-level program designs into the higher-level design views. Well understood rules

can be encoded into a reverse engineering tool, while others must be solicited from a human expert. Many inferences cannot be done automatically, as the relevant information (design decisions and application domain knowledge, for example) are not explicit in code. This makes the involvement of a domain expert critical in any but a trivial reverse engineering task. We assume close co-operation between a domain expert and a knowledge-based reverse engineering assistant tool. Reverse engineering progresses in steps. At each step, a tool applies heuristic rules to extract design views, while a domain expert accepts/rejects decisions made by a tool. The above observations are reflected in an incremental and semi-automatic reverse engineering process model in Figure 2.

We do not set any a priori border between the automated and human-guided part of the reverse engineering process. The level of possible automation differs across reverse engineering projects and is only determined by one's understanding of the reverse engineering heuristics involved in a given project. The more we understand, the more we can automate. The heuristics that we understand, to the extent that we can specify them in a precise way, can be automated.

Refronte, a front-end of a reverse engineering tool, parses sources into a program knowledge base, PKB for short. The PKB contains program design abstractions that can be automatically computed from sources using traditional compilation techniques. Typical examples are control flow graphs, attribute syntax trees, procedure calling graphs, etc. The actual types of design abstractions that we compute depend on the specific objectives of a reverse engineering project. Bubbles 'Reverse 1' and 'Reverse 2' in Figure 2 represent applications of reverse engineering heuristic rules in an automatic, semi-automatic or manual manner. Icons labelled with 'DKB-i' represent intermediate design views obtained during reverse engineering. A human expert (data analyst, programmer or application domain expert) inspects intermediate design views and can influence the reverse engineering process. Dotted lines in Figure 2 indicate involvement of the domain expert in the reverse engineering process.

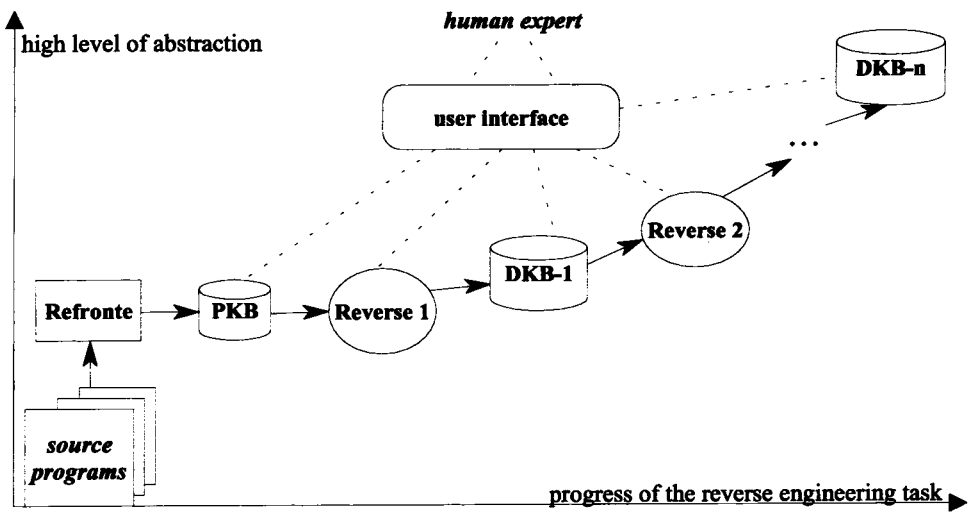


Figure 2. Semi-automatic and incremental reverse engineering process

3.2. The basis for model-based reverse engineering tools

We wanted to find a uniform way of specifying what a reverse engineering tool does, independently of many variations involved in reverse engineering projects. Three elements describing capabilities of reverse engineering tools are low-level program design models (PKB in Figure 2), higher-level design models (DKBs in Figure 2) and reverse engineering heuristics. We developed notations to conceptually model program design at all abstraction levels and a notation for specifying reverse engineering heuristics in terms of conceptual program design models. These notations capture essential commonalities of reverse engineering tools and facilitated model-based design of a generic design abstractor. In another source (Jarzabek, 1998), we described a program query language (PQL), a formalism to specify questions about properties of programs. A *PQL* interpreter automatically answers questions, helping programmers in analysis of programs for understanding during software maintenance and re-engineering. We extended *PQL* with new features that allowed us to specify how higher-level design models are to be produced based on analysis of low-level design patterns. Extended *PQL* is a declarative language, suitable for describing reverse engineering heuristics.

Two further arrangements underlie the model-based design of a generic design abstractor. Firstly, we decided to separate low-level program design models (stored in the PKB, Figure 2) from the actual mechanism used to compute these models (i.e., *Refronte* in Figure 2). In general, any program information that can be computed using conventional compilation techniques can be included in the PKB. The actual contents of the PKB depend on the source language and on the class of reverse engineering heuristics we wish to describe. In our approach, proper reverse engineering starts with the PKB. This is not to say that computation of low-level design abstractions from sources is always easy or not relevant to reverse engineering. On the contrary, it is important and it does pose its own problems. However, such a separation of concerns simplifies greatly the description of essential reverse engineering activities and allows us to view reverse engineering uniformly as mappings between program design models. The Refronte deals with the source code.

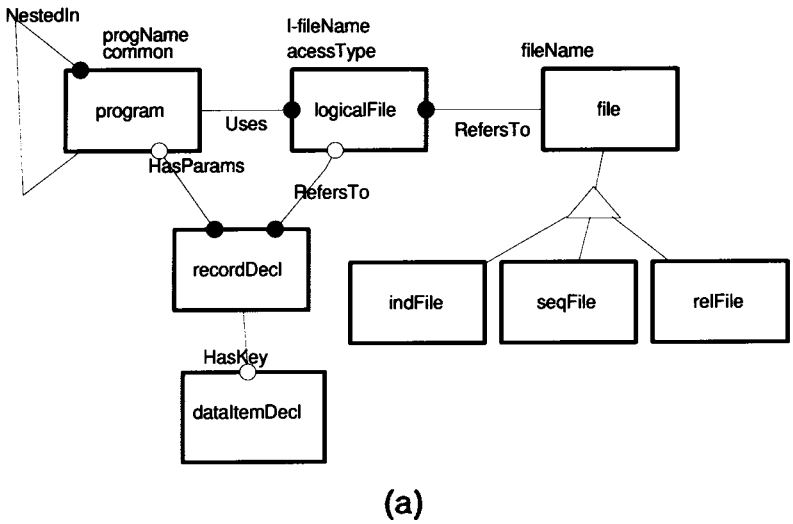
Secondly, we separated conceptual models of program design from the physical storage for program designs in the PKB and DKBs. Again, the issue of storing the program design information is by no means a trivial one. Possible media to store program information range from tree structures, to relational databases (Linton, 1984), PROLOG, object-orientated databases (Burson, Kotik and Markosian, 1990), program dependency graphs (Ottenstein and Ottenstein, 1984) and hybrid program representations (Jarzabek, Shen and Chan, 1994). We refer the reader to other publications for details. The choice of the right storage medium depends on the program information we wish to store and on the ways we wish to manipulate, query and transform this information. Conceptual models of program design make it possible to design a design abstractor independently of the physical storage for program designs.

4. CONCEPTUAL MODELS OF PROGRAM DESIGN

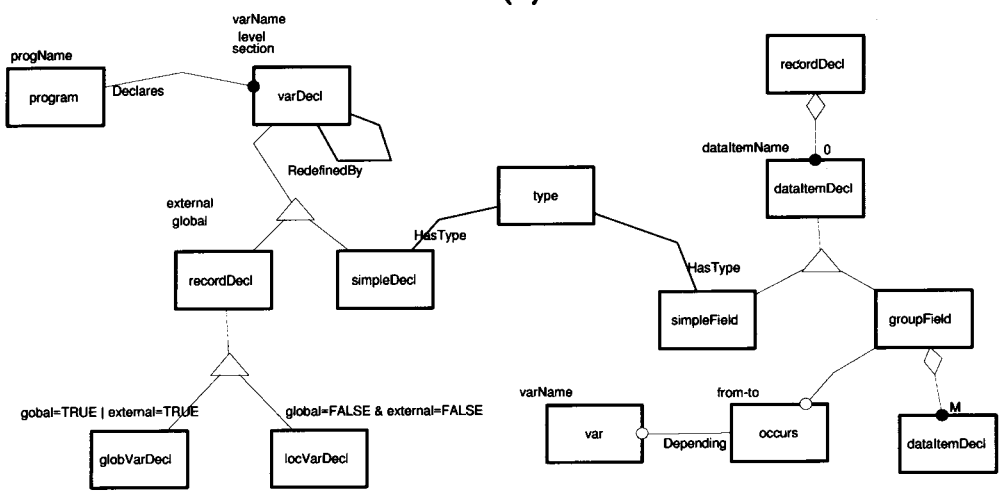
In *PQL*, we can define views of the program designs stored in the PKB or DKB. A *PQL* query is written in terms of conceptual models of program design information stored

in the PKBs or DKBs. In this section, we explain how we conceptually model program design. For example, Figures 3(a) and 3(b) depict parts of the PKB model that we used in recovering data models from COBOL85 programs.

The program design models of Figure 3 are created using the OMT notation (Rumbaugh *et al.*, 1991). We believe models are intuitive and we only briefly comment on them. Program models are expressed in terms of program entities (in rectangular boxes), their attributes (displayed above the entity box) and entity relationships. The meaning of a relationship link is clarified by a name attached to a link. A filled circle at the end of a relationship link stands for 'many' connectivities, while an open circle stands for '0 or



(a)



(b)

Figure 3. An excerpt from the COBOL85 PKB model: (a) global relations; (b) data descriptions

1' connectivity. Design entities are classified using an *IsA* relationship (a triangle with a general entity above and specialized entities below). Relationships defined for a parent entity (e.g., procedure) apply to child entities. Attributes assigned to a parent are inherited by all its children. In textual form, we refer to entity attributes using dot notation, e.g., *program.progName*. We refer to relationships using predicate notation, e.g., *Uses (program.logicalFile)*.

From Figure 3(a) we read that a COBOL85 program may use many files. While there can be only one physical name for a file in the operating system, a physical file can be referred to using different logical names in programs. Entities *logicalFile* and *file* model this situation. Links between logical files and physical files are defined within a program, in an ASSIGN clause of a FILE-CONTROL paragraph (sometimes links are established in JCL statements and interactive program set-up procedures). All these sources are used to establish relationship *RefersTo* between entities *logicFile* and *file*. Files can be organized into three categories: sequential, indexed and relative. This is modelled by *IsA* classification, represented by a triangle in diagrams. Indexed files have a corresponding key that is used to selectively read records from the files.

Diagrams of Figure 3(b) define the structure of syntax entities such as *recordDecl* and *dataItemDecl*. Syntax entities represent language constructs and correspond to abstract syntax grammar symbols. Syntax entities are also classified using an *IsA* relationship. Attributes (displayed above the syntax entity box) assigned to a parent apply to all its children. Attributes specify types of information that are available at syntax tree nodes.

Lists of elements are identified by aggregation one-to-many relationship links. There are two lists in Figure 3(b), namely, a list of data items in a record declaration (connectivity 0 above the *dataItemDecl* box indicates that this list may be empty) and a list data items in a sub-record (*groupField*). Syntax entity *occurs* is an optional component of a *groupField*. Entities in structure diagrams can be given roles, e.g., *Depending* is the role of entity *var* as a component of construct *occurs*.

The reader should observe that program design models are created for a reverse engineering project in hand and models will differ substantially across the projects. The 'knowledge contents' of the models depends on the source language, on what we are trying to achieve by means of reverse engineering and on types of reverse engineering heuristics to be applied. Our notations do not restrict the semantic contents of models, but, obviously, we should not model the information that we cannot compute. We do not have a descriptive definition of the formal semantics for program design models. The operational semantics of the design models is implicitly defined by *Refronte* actions (for low-level designs in the PKB) and by reverse engineering heuristics combined with human actions (for higher-level designs in DKBs).

5. AN OVERVIEW OF THE *PQL* NOTATION

For the reader's convenience, we shall summarize the *PQL* features that are essential to this paper. We refer the reader to another source for a comprehensive presentation of *PQL* (Jarzabek, 1998).

In *PQL*, each entity name (e.g., program) represents a set of its instances. The

relationship signature (e.g., Uses (program, logicalFile)) represents a subset of pairs of entity instances that are involved in a relationship.

We specify the query result (i.e., a view to be extracted) as a *tuple*. Tuples may involve entity instances as well as attribute values. Below, we refer to the program design models of Figures 3(a) and (b). Here are examples of tuples:

⟨program, logicalFile, file⟩—a set of triples whose first element is an instance of entity 'program', the second element is an instance of entity 'logicalFile' and the third one an instance of 'entity file',

⟨program, program.progName⟩—a set of pairs, the first of which is an instance of entity 'program' and the second one is a 'program name',

file—a set of instances of entity 'file' (a one element tuple),

file.fileName—a set file names.

A list is an entity type that refers to repeated syntax entities (e.g., dataItemDecl*—a list of data item declarations of 0 or more length).

Here are examples of simple queries:

Select program

Meaning: returns a set of all instances of entity 'program' from sources.

Select program.name

Meaning: returns a set of values of attribute 'name' for all instances of entity 'program'.

Select ⟨program, logicalFile,file⟩

Meaning: returns a set of all the triples whose first element is an instance of entity 'program', the second element is an instance of entity 'logicalFile' and the third one an instance of entity 'file'.

We can constrain properties of the target subset by specifying conditions to be satisfied by its elements.

Conditions are expressed in terms of:

(a) entity instances, attribute values and constants,
(b) participation of entities in relationships,
(c) code patterns.

The following is a format of a program query:

select-cl::=[name-cl] declaration* **Select** result cond-cl [from-cl][pattern-cl]*
cond-cl::=(with-cl | suchthat-cl)*

Clauses in rectangular brackets are optional. Star '*' means repetition 0 or more times. Stepping from left, the *name*-cl gives a name to a retrieved program view. Declarations introduce synonyms for entities. Synonyms can be used in the remaining part of the query to mean a corresponding entity. The *result* specifies a program view to be produced. A program view is a subset of entity instances, a subset of tuples or entity attribute values. In the *with*-cl we constrain attribute values (e.g., procedure.procName = 'Edit-Order').

The *suchthat*-cl specifies conditions in terms of relationship participation. The *pattern*–cl specifies patterns in terms of syntax structure definition (e.g., *recordDecl* in Figure 3(b)).

Here are examples of two program view definitions expressed in *PQL*:

Find programs that use logical file 'CustomerFile':
Select program **such that** Uses (program,logicalFile) **with** logicalFile.l-fileName= 'CustomerFile'
Find sub-records that contain a field named 'rate':
Select groupField **pattern** groupField (_,_dataItemDecl_) **with** dataItemDecl.dataIt - emName = 'rate'
Explanation: here we are looking for syntax tree patterns with root node 'groupField' whose left child is not constrained and the right child (a list according to syntax definition in Figure 3(b)) contains 'dataItemDecl' named 'rate' on any position of the list.

To further illustrate the use of *PQL*, we shall now specify two reverse engineering heuristics for recovering entity-relationship (ER) data models from COBOL85 file structure definitions. Below, the reader will find informal descriptions of heuristics followed by their definitions in *PQL*. *PQL* expressions refer to the program design models of Figures 3(a) and (b).

R1. Finding candidate entities for the ER data model. Candidate entities correspond to files and sub-records with more than one record field.

Specification of candidate entities in PQL:
file, dataItemDecl *IsA* candidate
view file-ent
Select file
Explanation: this view identifies candidate ER entities that correspond to physical files in a COBOL application. The first line says that, name 'candidate' will denote either entity 'file' (Figure 3(a)) or 'dataItemDecl' (Figure 3(b)). Name 'candidate' can be used in all view definitions below. The second line gives name 'file-ent' to this view.
view group-ent
dataItemDecl+ dataItems
Select groupField **such that**
Parent* (recordDecl, groupField) **and** recordDecl.section = File-Section **and** groupField (dataItems) **and** LENGTH (dataItems) $>$ 1
Explanation: this view identifies candidate ER entities that correspond to sub-records including more than one field. The second line introduces name 'dataItems' as a synonym for a non-empty list of entities 'dataItemDecl'. Then, we select all sub-records of length more than one from records declared in the file section of programs under consideration.
view entities
Select candidate **such that** IS-IN (candidate, file-ent) **or** IS-IN (candidate, group-ent)
Explanation: this view defines a set of candidate ER entities as a union of views

'file-ent' and 'group-ent'. This feature allows us to build views in a hierarchical way.

R2. Finding entity relationships. Foreign keys embedded in records indicate a possible relationship between entities representing corresponding files. Furthermore, a one-to-many relationship occurs between an entity containing a repeated group of record fields and the entity that represents this group. Foreign keys are those record fields that have the same name as a key of some other record. Both records must be connected to a COBOL file, therefore, they should appear in the File Section of a COBOL program.

Specification of candidate entity relationships in PQL:
view foreign-keys
recordDecl rec1, rec2
simpleField foreignKey, key
Select <rec1, foreignKey, rec2> **such that** Parent* (rec1, foreignKey)
with rec1.section = 'File-Section'
such that exists [key **such that exists** [rec2 **with** rec1.section = 'File-Section'
such that HasKey (rec2, key) **and** rec1 ≠ rec2]
with key.dataItemName = foreignKey.dataItemName]

Explanation: The second and third lines declare synonyms for entities 'recordDecl' and 'simpleField'. Then, we consider records declared in file sections (those are the records that possibly correspond to ER entities) and select those record fields whose names are identical to other record fields that are known to be keys. We do not consider composite keys in this example.

Candidate entities from the view obtained by applying rule R1 should be presented to a data analyst. A data analyst can further explore the PKB using *PQL* queries to determine which files represent true domain concepts and should be selected as entities for the ER data model. Similarly, a data analyst should analyse the information selected in 'foreign-key' view to exclude accidentally identical record field names as possible foreign keys. True foreign keys would then be interpreted as relationships between entities. After these steps, the first-cut ER data model is generated.

6. RECOVERING OBJECTS FROM PROCEDURAL CODE

In this section, we shall further illustrate our modelling method with examples from object recovery. Like reverse engineering of data, techniques for recovering object-orientated views from procedural programs have received much attention because of market need. Many companies consider adopting an object-orientated approach, but the question remains what to do with legacy software written in procedural languages. One possible solution is to gradually re-engineer legacy software into object-orientated architecture (Jackobson and Lindstrom, 1991). Another context in which techniques for recovering object-orientated views are promising is reuse re-engineering (Canfora, Cimitile and Munro, 1994).

In procedural programs, objects are implemented by data structures and fragments of procedural code that manipulate these data structures. In well-designed programs, each of

the object's operations might be already implemented as a separate procedure. In poorly-designed programs, code implementing operations may be delocalized and intermixed with implementations of other program functionalities. In such cases, finding objects will involve slicing programs in order to isolate code that implements the object's operations from the rest of a program (Weiser, 1984). If objects have not been clearly identified at the design phases, recognizing objects is difficult.

Liu and Wilde (1990) proposed two methods for identifying objects in code. The first method starts by identifying global data structures that are likely to represent the state of some object. Then, clusters of procedures that refer to global data are identified and relationships between such clusters are analysed to find object's operations. The second method starts by identifying dependencies between data types in a program and then one groups together procedures whose headings share common data types to form objects.

Canfora, Cimitile and Munro (1994) classify object-like, possibly reusable, abstractions into four groups: abstract objects (data structures + operations), abstract data types (data types + operations + instantiation), generic data structures (abstract objects parametrized by types for generality) and generic abstract data types. Then, the authors identify candidature criteria for each group ('candidature criterion' closely corresponds to our notion of 'reverse engineering heuristic'). In their other paper (Canfora *et al.*, 1994), the authors describe a logic-based notation for specifying candidature criteria and the program information required to isolate candidates. Gall and Klösch (1995) describe yet another approach to finding objects. They start by reverse engineering structure charts and data flow diagrams. Then, they build up objects around data store entities and around other non-persistent, but important from the application domain point of view, pieces of information. Sneed and Nyary (1995) describe many types of objects that may be of interest in re-engineering of data processing applications. Object types include user interface objects, data objects, file objects, view objects and many others. In conclusion, depending on the type of the subject application, project objectives, and types of objects we are interested in, we must identify and apply proper heuristics for object recovery.

Suppose we are re-engineering C programs into C++ and we wish to identify candidate classes in C code. The process that leads to identifying classes is semi-automatic and involves applying heuristics (such as data coupling between C functions) and manual analysis of programs by a human expert. A domain expert is aware of application domain concepts that are candidates for classes while the software engineering expert can judge which design and implementation concepts are candidates for sound and useful classes. Automatically identified classes must be presented to human experts for acceptance and refinement.

6.1. Object recovery heuristics

Rules listed below are based on object identification heuristics proposed by others (Canfora, Cimitile and Munro, 1994; Liu and Wilde, 1990; Gall and Klösch, 1995):

H1. Any global data structure, say data X, directly referred to (used or modified) by two or more functions should be considered a candidate data rep (or part of it) for some object X.

H2. Any function that refers to data X should be considered a candidate method for object X.
H3. Any two candidate data reps, say data X and data Y, identified by applying rule H1, and such that there is a function that refers to both data X and data Y, are likely to form together a data rep for the same object. Candidate methods for this object are functions that refer to any of the data that form the object's data rep.
H4. Any user-defind data type, say type X, that appears in the headings of two or more functions, such that the headings of those functions do not include any supertype of type X, should be considered a candidate data rep (or part of it) for some class X.
H5. Any function whose heading includes type X should be considered a candidate method for class X.
H6. Any two candidate data reps, say type X and type Y, identified by applying rule H4, and such that there is a function whose heading includes both type X and type Y, are likely to form together a data rep for the same class. Candidate methods for this class are functions whose headings include any of the data types that form the data rep of the class.

Notice that we consider both abstract objects (global data structures + operations) and ADTs (user-defined data types + operations + instantiation) as candidates for C++ classes.

6.2. A plan for semi-automatic recovery of abstract objects as candidates for classes

To see how the above heuristics could be used, we shall outline a possible plan for semi-automatic recovery of candidate classes. First, we focus on abstract objects.

O1. Apply rule H1 to find candidate data reps for objects.
O2. Let the human expert select one of the candidate data reps, say data X, and decide whether or not it forms a basis for a sound design or application domain object. To help in making the decision, the expert will view functions that refer to data X (rule H2).
O3. If the expert decides that there are not enough reasons to further evaluate data X as a data rep for some object, he/she will repeat step O2 for other data; else, the expert will proceed to step O4.
O4. Apply repeatedly rule H3 to find other data that, together with data X, are likely to form a data rep for object X. To help in making the decision, the expert will view functions that refer to candidate data reps (rule H3).
O5. At this point, the expert has already identified a set of data that, together with data X, form a data rep for object X.
O6. Find candidate methods for object X (rules H2 and H3). Let the human expert view and verify the candidate methods. This ends recovery of object X.
O7. Repeat steps O2–O6 until all the candidate data reps have been considered.
O8. Map recovered objects to classes.

6.3. A plan for semi-automatic recovery of ADTs as candidates for classes

A1. Apply rule H4 to find user-defined types that may be candidates for ADTs.
A2. Let the human expert select one of the candidate types, say type X, and decide whether or not it forms a basis for a sound design or application domain ADT. To help in making the decision, the expert will view functions whose headings include type X (rule H4).
A3. If the expert decides that there are not enough reasons to further evaluate type X as a candidate for ADT, he/she will repeat step A2 for another data type; else, the expert will proceed to step A4.
A4. Apply repeatedly rule H6 to find other data types that, together with type X, are likely to form an ADT X. To help in making the decision, the expert will view functions whose headings include data types under consideration (rule H6).
A5. At this point, the expert already has identified a set of data types that, together with type X, form an ADT X.
A6. Find candidate methods for ADT X (rules H5 and H6). Let the human expert view and verify the candidate methods. This ends recovery of ADT X.
A7. Repeat steps A2–A6 until all the candidate type reps have been considered.
A8. Map recovered ADTs to classes.

6.4. Specifying object recovery heuristics in PQL

We start by modelling those aspects of C and C++ program designs that are relevant to object recovery. As notions of a class, method and data representation for objects are not present in the domain of C programs, we need to model objects independently of C programs. A PKB (program knowledge base) is a repository that stores source program design and DKB (design knowledge base) is a repository that stores reverse engineered design views (in our case, object models). Figure 4 depicts conceptual models of the PKB and DKB.

With reference to the program models of Figure 4, we shall now define object recovery heuristics in *PQL*. In all the definitions, we use names *data*, *data-X* and *data1* as synonyms of design entity *globData* and name *fun* as a synonym of design entity *Cfunction*. The following *PQL* declarations introduce synonyms:

globData data, data-X, data1
Cfunction fun
The definition of first-cut candidate objects (heuristics H1):
view objects
Select <data,fun> **such that** Refers (_,data>1) **and** RefersTo (fun,data)
Explanation: underscore means an unbound variable.

The definition of first-cut candidate methods (heuristics H2):

view methods
Select <fun> **such that** <_,fun> ∈ objects

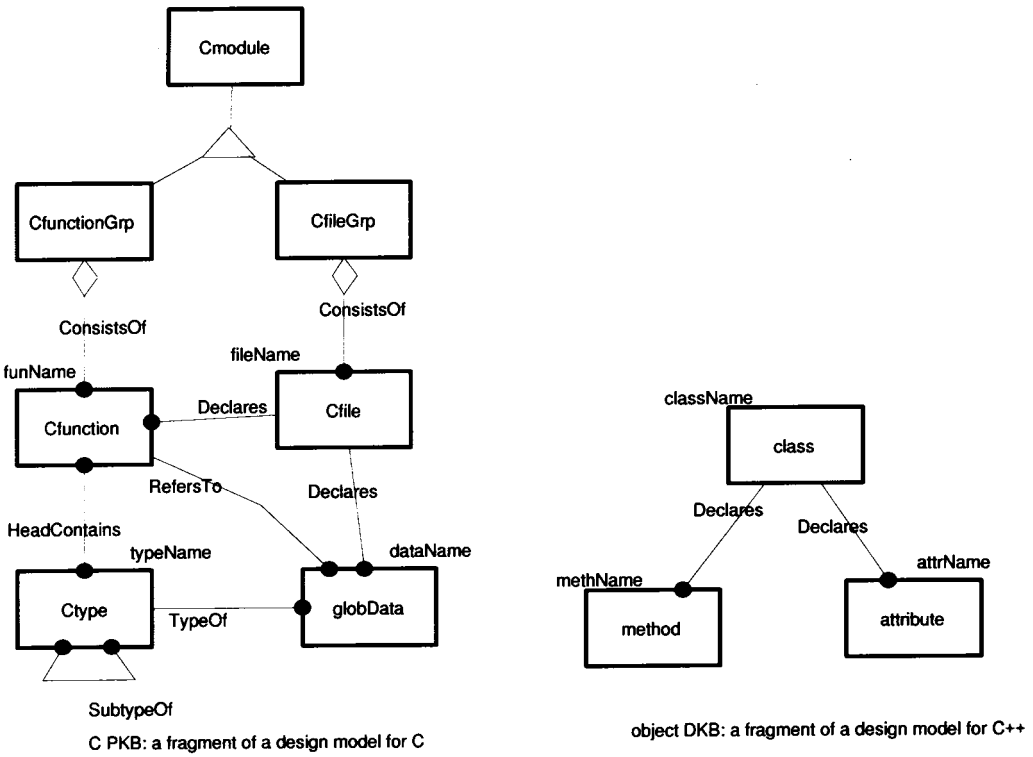


Figure 4. Design models for C PKB and object model DKB

The definition of first-cut candidate data reps for objects (heuristics H3):
view data-rep
Select <data> **such that** <data,_> ∈ objects
Explanation: this view is presented to the human expert in steps O1 and O2.

For a selected data X from the set data rep, we now produce first-cut methods for a candidate object X with data rep data X (rule H2):

view method-X-1
Select <fun> **from** methods **such that** RefersTo (fun,data-X)
Explanation: this view is presented to the human expert in steps O2 and O3.

view data-rep-X
Select <data> **such that** data = data-X
Explanation: data X is judged to be a data rep for a sound object.

For a selected data1 from set data-rep-X, we find other data that, together with data X, form a data rep for the same object:

view data-rep-X-more
Select <data> **from** date-rep **such that** data ∉ data-rep-X
and such that exists [fun **in** methods **such that**
RefersTo (fun, data1) **and** RefersTo (fun, data)]
Explanation: data from set data-rep-X-more are presented to the expert who will decide whether the data should be included into the set data-rep-X. This view is computed repeatedly as long as new data are contributed to the set data-rep-X (step O4 and rule H3).

The alternative solution to computing the data rep for object X might be to compute all the candidate data reps of object X and then ask the expert to review the candidates and decide which ones should collectively form a data rep for object X. Computing candidate data reps is formalized in the following recursive view:

recursive view data-rep-X
Select <data> **from** data-rep **such that**
exists [fun **in** methods **such that**
exists [data1 **in** data-rep-X **such that** RefersTo (fun, data1) **and**
RefersTo (fun, data)]]
Explanation: here, we find other data that, together with data X, possibly form a data rep for object X, based on rule H3. A recursive view is executed repeatedly as long as new data are inserted into the set data-rep-X. The reader can find out about strategies for evaluating recursive queries in Gardarin and Valduriez (1989). Our current implementation does not include recursive views.

The definition of methods for object X (heuristic H3):

view methods-X
Select ⟨fun⟩ **from** methods **such that exists** [data **in** data-rep-X
such that RefersTo (fun, data)]
Explanation: the expert reviews and verifies candidate methods (step O6).

This completes recovery of design views relevant to object X as a candidate class for a C++ program. We do not formalize heuristics for recovery of ADTs as they are similar to those described above.

6.5. Generation of recovered design views in extended PQL

In the above discussion, we have recovered design views that correspond to elements in the object model but we have not created object models in an explicit way. We have extended *PQL* to allow for generating design fragments using terminology of the target design, in the case of our example, object-orientated design. Generation rules have the following format:

source-pattern ⇒ (**CreateEnt** entitySpec |
CreateRel relSpec |
AssignAttr attrValueSpec |
CreateLink linkedEntitiesSpec)*

The 'source-pattern' is a pattern definition in *PQL*. The right-hand side of the formula shows how to create fragments of the reverse engineered target design when instances of the source-pattern are found in the PKB. Operator **CreatEnt** creates a new instance of a design entity. **CreateRel** establishes a relationship between two design entities. **AssignAttr** assigns a value to the entity or relationship attribute and **CreateLink** links design entities from different program design models for the purpose of traceability. Operators may appear any number of times and in any order on the right-hand side of the formula.

In the object recovery example, the 'source-pattern' refers to C models (on the left-hand side of Figure 4), while target design entities are classes, attributes and methods of the object-orientated design model (on the right-hand side of Figure 4). In the extended *PQL*, we can map data from set data-rep-X to class attributes and functions from set methods X to class methods:

CreateEnt class **with** class.className = ?X
Explanation: ?X indicates that the class name must be provided by the human expert.
Select fun **from** methods-X
CreateEnt method **with** method.methName = fun.funName
CreateRel Declares (class, method) **with** class.className = ?X
CreateLink (fun, method)
Select data **from** data-rep-X
CreateEnt attribute **with** attribute.attrName = data.dataName
CreateRel Declares (class, attribute) **with** class.className = ?X
CreateLink (data, attribute)

7. REVERSE ENGINEERING TOOL ARCHITECTURE

Figure 5 depicts major components of our reverse engineering tool.

- ***Refronte*** (*R*everse *E*ngineering *FRONTE*nd)

The *Refronte* extracts low-level program design information from sources and puts them into the program knowledge base (PKB).

- ***Design abstractor***

The design abstractor evaluates reverse engineering heuristics. The design abstractor understands conceptual program design models and heuristics written in extended *PQL*. Mappings of conceptual models of PKB and DKBs to physical schema allow the design abstractor to evaluate heuristics. Recovered design models are stored in design knowledge bases (DKBs).

- ***User interface***

A user interface allows a human expert to interact with the reverse engineering process. Through the user interface, a human expert can extend/modify built-in reverse engineering heuristics and examine intermediate program designs.

- ***Export–import facility***

The final program design abstractions can be transferred from the DKB into other presentation and analysis tools, for example into a repository of a CASE tool. The *export–*

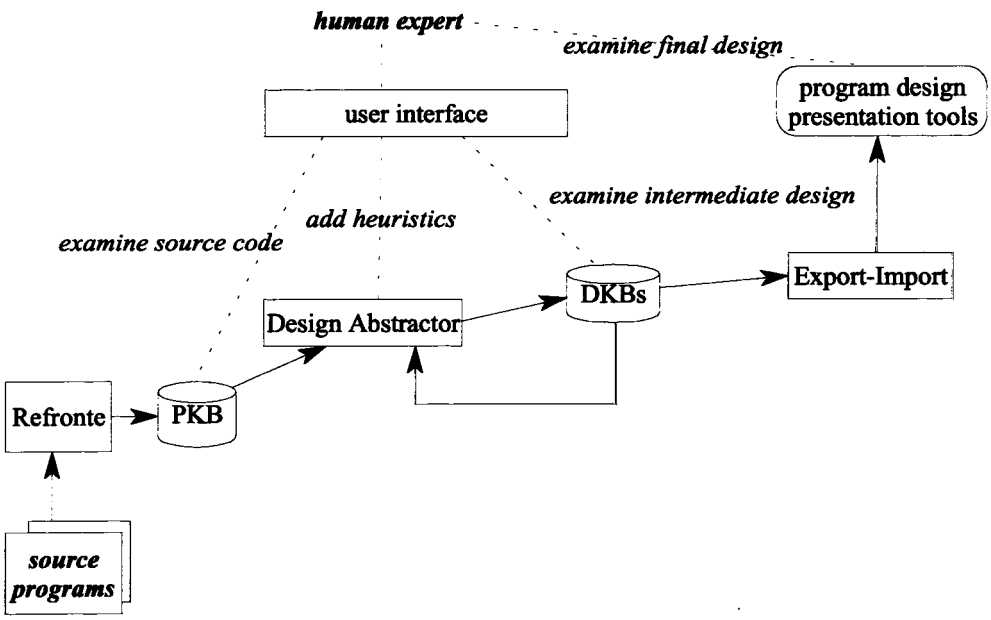


Figure 5. Components of a reverse engineering tool

import facility translates the design views from the DKB format into the format required by the target design presentation tool. In our experiments, we imported reverse engineered design views into POSE[1] diagramming editors. A programmer can use POSE editors to further refine and process design views.

8. DESIGN OF A MODEL-BASED GENERIC DESIGN ABSTRACTOR

In Section 3, we discussed two design decisions that are instrumental in the model-based design of reverse engineering tools, namely, separation of low-level program design models from the actual mechanism used to compute these models, and separation of conceptual program design models from the physical storage for program designs in the PKB and DKBs. These two decisions ensure genericity and flexibility of the reverse engineering tool. *The first design decision* allows us to uniformly view all types of reverse engineering heuristics as transformations between source and target program design models. With standard notations for describing program design models and for writing reverse engineering heuristics, we can design a generic design abstractor, parametrized by (variant) source and target program design models. *The second design decision* allows us to deal with another important variant requirement for reverse engineering tools, namely, the physical storage for program designs. As reverse engineering heuristics are expressed in terms of

[1] POSE™ by CSA Research Pte Ltd.

the conceptual program design models, we can implement the interpreter of reverse engineering heuristics driven by conceptual design models and *PQL*.

A *PQL* interpreter forms the core of a generic design abstractor. Of course, to evaluate reverse engineering heuristics, the design abstractor must 'know' how to relate conceptual program designs to their physical representations. For example, if a reverse engineering heuristic refers to the control flow relation between two program fragments, then the design abstractor must get the control flow information from the physical program design storage. There are many ways how we can represent program control flow information and, to be generic, the design abstractor should not make any assumptions whether the information is stored in an annotated syntax tree, a dependency graph or, perhaps, is not stored at all but computed on demand (Reps, 1994). To solve this problem, the design abstractor must maintain mappings between elements of the conceptual design and their physical counterparts. Below, we describe in more detail how we implemented these concepts within a generic design abstractor.

8.1. The first prototype

The parameters that drive a generic design abstractor are definitions of conceptual models for the source and target designs and mappings between conceptual design models and their physical representation (Figure 6). By feeding the parameters, we obtain a design abstractor customized to specific source and target design models and to the specific physical storage for program designs. While it is useful to have a library of pre-defined heuristics, design abstractors obtained from the generic architecture are flexible in the sense that they can evaluate any new heuristic written in *PQL*.

We store definitions of the source and target design models in five tables describing design entities, entity relationships, attributes of entities and relationships, *IsA* classification of entities and abstract syntax rules, respectively. These five tables isolate and hide the knowledge of source and target design models, making the computational core of the design abstractor independent of the specific models involved.

Now we shall describe how we isolated our design abstractor from the physical storage for program designs. In the first prototype, we stored both the source and target designs in PROLOG. Figure 7 depicts modelling schema and Figure 8 – excerpts from the COBOL model.

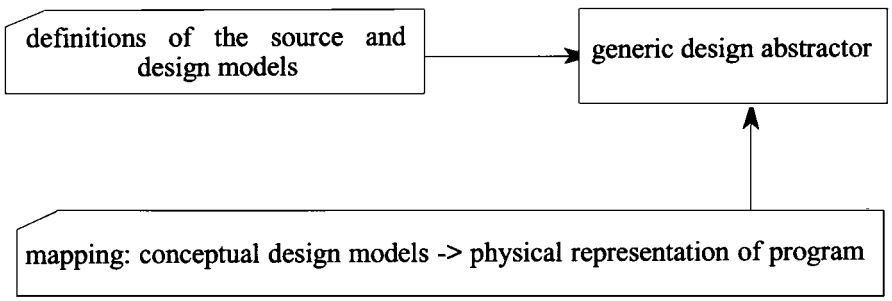


Figure 6. Parameters driving a generic design abstractor


```
IsEntity (entName)
IsEntRelationship (relName, entName1, entName2)
IsAttribute (entName, attrName, valueType)
IsRelAttribute (relName, attrName, valueType)
IsA (entName1, entName2)
IsFirstChild (parentEntName, childEntName, type)
IsRightSibiling (leftEntName, rightEntName, type)
```

Figure 7. The schema for a conceptual program design models in PROLOG

```
IsEntity (program), IsEntity (logicalFile)
IsAttribute (program, progName, String)
IsA (simpleDecl, varDecl)
IsFirstChild (recordDecl, dataItemDecl, list)
```

Figure 8. Excerpts from the conceptual model definitions of Figures 3(a) and (b)

The first predicate in Figure 9 defines instances of various design entities; the second one—interrelated entity instances, the third and fourth ones—values of entity and relationship attributes, the fifth and sixth ones—the shape of abstract syntax trees. Translation of reverse engineering heuristics expressed conceptually in *PQL* into equivalent heuristics written in PROLOG was also straightforward. This implementation had, however, two major shortcomings. Firstly, we did not really factor the knowledge of the physical storage for program designs from the design abstractor. In particular, mappings between conceptual and physical program designs were implicit, encoded in the design abstractor. Therefore, our design abstractor could not be easily ported to other physical representations of program designs. Secondly, our PROLOG solution was very inefficient. While it is rational to store global program design information (such as procedure call trees or global data usage information) in a PROLOG database, storing the detailed program design information, such as annotated syntax trees, in PROLOG is not practical. Evaluation of reverse engineering heuristics that involve traversing tree- or graph-like structures leads to serious performance problems. Our implementation of the design abstractor on the PROLOG had

```
1. Entity (entName, entInst)
2. Relationship (relName, entInst1, entInst2)
3. EntAttribute (entInst, attrName, val)
4. RelAttribute (relName, entInst1, entInst2, attrName, val)
5. FirstChild (ent1, ent2)
6. RightSibling (ent1, ent2)
```

Figure 9. Predicates used for instantiating source and target design models

a value as a proof-of-concept but we wanted to find a general solution to a generic design abstractor problem and one that would offer better options for tuning the performance than PROLOG did.

8.2. Refinement of a generic design abstractor

We refined the first prototype by isolating the knowledge of a physical storage for program designs from the design abstractor. The main problem was to let the design abstractor manipulate physical designs via conceptual models of the designs. At the beginning we opted for descriptive specifications of mappings between conceptual and physical designs. We hoped to find mappings that would facilitate automatic generation of operations to access any physical storage of program designs that one might want to use. The design abstractor would call the access functions during evaluation of reverse engineering heuristics. However, we failed to find a formalism that would be general enough to encompass any possible physical storage. We were also afraid that descriptive specifications of mappings may create yet another bottleneck for the performance. Therefore, we decided to define mappings in an operational way. We identified abstract interface operations that extract the design information from the physical program design storage. We implemented the generic design abstractor in terms of this interface. The interface operations hide the details of the physical storage for program designs from the design abstractor. Whenever we want to port a design abstractor to a new storage for program designs, we re-implement the interface operations in terms of the new representation. This approach does not impose any performance penalty and is flexible: the design abstractor does not have to know whether a given design information is pre-computed and permanently stored or computed on demand. This knowledge is encapsulated in the interface operations. Unified modelling conventions helped us again to identify a fixed set of abstract (i.e., source and target design-independent) interface operations that are sufficient for a design abstractor to evaluate reverse engineering heuristics (Figure 10). The signatures of the interface operations do not change when we change source and target design models or when we wish to use another physical storage for program designs. In the latter case, of course, we must re-implement the interface operations.

8.3. How does a design abstractor work?

The design abstractor starts by parsing a reverse engineering heuristic into an internal form (Figure 11).

We represent reverse engineering heuristics as abstract syntax trees. The heuristic parser breaks a reverse engineering heuristic tree into specification fragments contained in the **from, such that, with** and **pattern** clauses. Specification fragments are interpreted in turn, dividing the heuristic evaluation process into a sequence of simpler evaluation steps. Evaluation of each heuristic specification fragment involves calls to one or more interface operations to access information from the physical storage for program designs. Evaluation of each fragment produces an intermediate result that better approximates the target design model specified by a reverse engineering heuristic. In the current implementation, we did not optimize the heuristic evaluation, but incremental heuristic evaluation offers ample


```
entInstSet GetEntInst (entName)              // returns a set of instances of entity entName
val GetAttrVal (entInst, attrName)           // returns the value of an attribute attrname of entInst
entInstSet GetRelated (relName, entInst1, _) // returns instances related to entInst1 by
                                             // relationship relName
entInstSet GetRelated (relName, _ , entInst2)
Bool IsRelated (relName, entInst1, entInst2) // returns true if entInst1 is related to entInst2 by
                                             // means of relationship relName
entInst GetParent (entInst)                  // returns the parent of entInst in the tree
entInst GetFirstChild (entInst)
entInst GetRightSibiling (entInst)
entInst CreateEntInst (entName)              // creates and returns a new instance of entName
CreateRel (relName, entInst1, entInst2)      // relates entInst1 to entInst2 by relName
SetAttrVal (entInst, attrName, val)          // sets value of attrName to val
```

Figure 10. Excerpts from the abstract interface to a physical storage for program designs

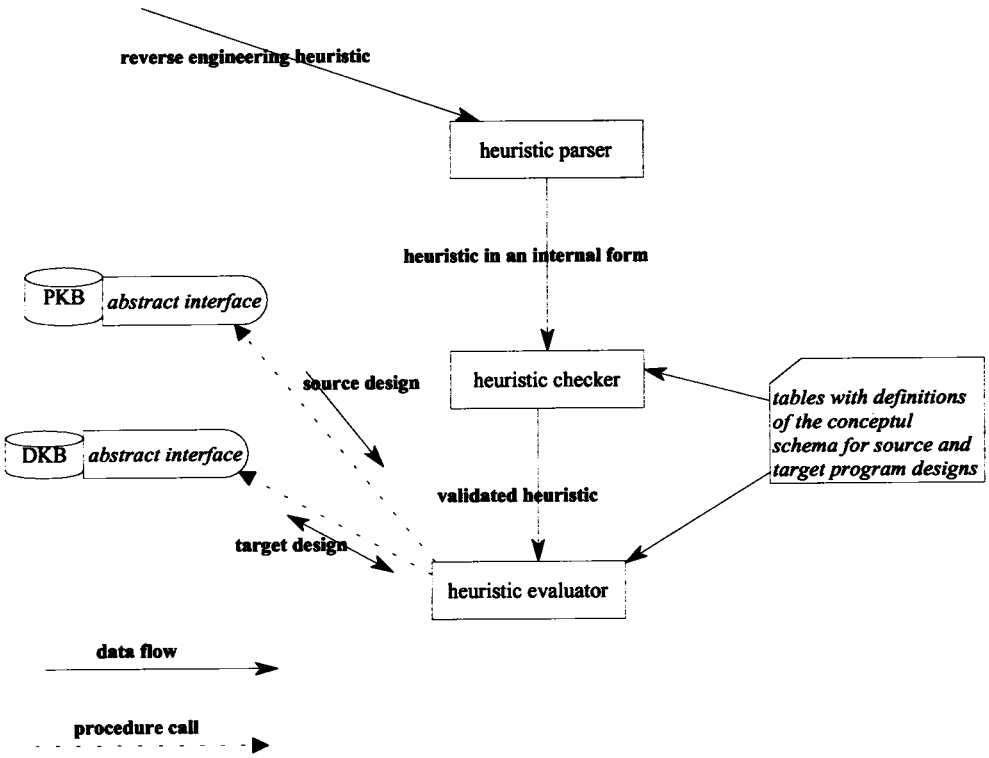


Figure 11. An architecture of a design abstractor

room for experimenting with various optimization techniques: we can change the criteria for identifying specification fragments without affecting the overall evaluation strategy. At the same time, the first-cut implementation of an unoptimized heuristic evaluator is relatively simple. We implemented a heuristic parser with a compiler-compiler LADE[2]. Other components of the design abstractor are implemented in C++ and PROLOG. We refer the reader to Jarzabek (1998) for a more detailed discussion of evaluation of queries expressed in descriptive languages such as *PQL*.

8.4. Customization of a generic design abstractor

To customize a generic design abstractor to a given reverse engineering task, we fill the tables with the schema for conceptual program design models (tables correspond to PROLOG schema in Figure 8) and implement the interface operations to a physical storage for program designs. We customize a generic design abstractor using the following steps:

(1) Decide what types of design views will be reverse engineered from sources.
(2) Identify reverse engineering heuristics that lead to producing required design views.
(3) Create conceptual models for source and target designs. Encode model definitions into the five tables.
(4) Decide upon the physical storage for program designs in the PKB and DKBs.
(5) Implement the abstract interface operations in terms of the selected physical storage for program designs.
(6) Include tables with conceptual design models into the generic design abstractor.
(7) Use a compiler generator to implement the *Refronte* to parse source programs. Write the PKB generation actions in terms of conceptual models, addressing only the program design information to be permanently stored in the PKB.
(8) Build a bridge to convert design information from the DKB into the format required by an import facility of a design presentation tool.

9. COMPARISON OF OUR SYSTEM WITH OTHER SIMILAR SYSTEMS

Having described our notations and design of a generic design abstractor, it is interesting to compare our system with other similar systems. Canfora, Cimitile and de Carlini (1992) describe the logic-based approach to the design of reverse engineering tools. Their tool stores design abstractions (namely, inter-modular data flow information) in a PROLOG database. Design views are then specified as PROLOG queries. The logic-based approach provides a flexible mechanism for filtering the design information extracted from code.

Work on program design query languages for specifying abstract program views is very much related to the model-based design approach described in this paper. Systems that have been designed with similar objectives to ours are REFINE (Burson, Kotik and Markosian, 1990) and SCA (Paul and Prakash, 1996). The three systems (REFINE, SCA and *PQL*) offer descriptive notations for specifying patterns. Patterns can involve syntactic

[2] LADE is a trademark of Xorian Technologies.

structures constrained with global program design and static semantic information (such as control and data flow in a program). All three notations can handle atomic data types, composite object types (such as structured loop constructs or expressions), list objects and allow one to organize program objects into subtyping hierarchies with inheritance. Though the three approaches have similar objectives, they differ in the design rationale, underlying program models, the way patterns are formulated, expressive power and ease of use. REFINE uses an object-orientated database to store program designs. In REFINE, one writes queries as declaratively specified functions. Functions can be recursive. SCA models programs as many-sorted algebras and offers a set of algebraic operations to specify program patterns. Based on a sound mathematical foundation, SCA's queries are concise and elegant. SCA allows one to reason about formal properties of patterns and reverse engineering heuristics.

GENOA (Devanbu, 1992) provides specification methods and algorithms that allow us to extract program design information from the source code. In GENOA, new types of queries can be formulated without actually modifying the Refronte at the cost of a more complex query specification language. Unlike GENOA, *PQL* is not suitable for specifying low-level program analysis, such as deriving data flow relations or computations of program slices. We assume that the underlying program design models are semantically rich, so that *PQL* can concentrate on essential aspects of reverse engineering rather than on deriving the low-level program design information.

The rationale behind *PQL* is to facilitate design of generic tools such as a design abstractor. We put much weight on simplicity and implementation issues. We designed *PQL* to be similar to SQL. There are, however, significant differences. In a relational database, we store information in a collection of tables. An SQL query specifies set operations that are executed on these tables to yield the result. *PQL* is based on the entity-relationship concepts that are more meaningful and on a higher level than relations. Furthermore, *PQL* has operations for manipulating structured objects such as trees and graphs. Forcing the program design information into the relational table format, although possible (Linton, 1984), is not appropriate as tree- and graph-like structures cannot be efficiently processed in this form. In general, the quest for the right storage for software engineering artefacts is not over yet (Godart and Charoy, 1994) therefore ability to work with multiple storage media is an attractive property of any system that deals with such artefacts.

We believe query optimization techniques developed in relational databases can eventually be applied to reverse engineering heuristic evaluation yielding the high performance required in the analysis of big programs. Unfortunately, we cannot support this claim by experimental results as our work on *PQL* optimization is at an early stage.

10. CONCLUSION

The work described in this paper is an attempt to find a unified view of reverse engineering methods and tools. We have tried to answer the question of what is common to various reverse engineering methods and tools and what are the essential differences between them. We used a modelling approach to investigate the answers to these questions. First, we examined a reverse engineering process, the behaviour of programmers involved

in that process and the role of reverse engineering tools. Next, we developed a notation to conceptually model program design information at various abstraction levels. With a conceptual model for program designs, we could view reverse engineering heuristics as mappings between source and target design models. We then developed a notation to specify such mappings. Based on these findings, we developed a generic design abstractor, a tool that evaluates reverse engineering heuristics to derive higher-level design abstractions from the lower-level ones. The generic design abstractor is parametrized by specifications of source and target program design models. By changing the parameters, we can adapt the generic design abstractor to the reverse engineering task in hand. The generic design abstractor is implemented in terms of an abstract interface to a physical storage for program designs. In that way, during customization, we can adapt the design abstractor to work with any physical storage for program designs, such as a relational database, PROLOG, object-orientated database, abstract syntax trees or a hybrid storage. As reverse engineering methods are applied in many different contexts, such flexibility is a desirable property of reverse engineering tools. Reverse engineering heuristics are complicated and difficult to formalize. We found the ability to easily modify tool functionality and to interact with the tool in a flexible way essential in those cases.

Tools built in an *ad hoc* way are not usually flexible enough, as possible variations in tool functions have not been incorporated into the tool architecture to make future customizations possible. *Ad hoc* design practice does not lead to accumulation of the tool design know-how, makes it difficult to repeat successful solutions and slows down the process of understanding and improving tool design methods. The model-based design method described in this paper addresses the above problems by providing a tool architecture and a formalism that allow us to design tools in a systematic way. Certain tool components that are expensive to implement, particularly a design abstractor, can be reused in many different reverse engineering contexts. Finally, a formalism for expressing reverse engineering heuristics allows us to record the reverse engineering know-how, reuse heuristics that are known to work from one project to another and automate well understood heuristics by encoding them into the design abstractor. We believe the design method described in this paper has potential to add value to a reverse engineering tool.

In future work, we plan to extend the *PQL* interface to the reverse engineering tool. We feel that more user-friendly interfaces should be built on top of the *PQL*. To apply design solutions presented in this paper to big industrial programs, we will implement a hybrid PKB for efficient evaluation of reverse engineering heuristics. We also plan to identify more heuristics and encode them into reverse engineering tools that we build.

Acknowledgements

This research was sponsored by the National University of Singapore Research Grant RP950616. Thanks are due to Tan Poh Keam who implemented the data reverse engineering tool (Jarzabek and Tan, 1995) and Ding Xin who extended our reverse engineering tool with the program transformation features.

References

Arora, A. K. and Davis, K. H. (1985) ‘A methodology for translating a conventional file system into an entity-relationship model’, in: Chen, P. P. (Ed), *Entity Relationship Approach: the Use of ER Concepts in Knowledge Representation*, North Holland, Amsterdam, pp. 148–159.

Bigerstaff, T. (1989). 'Design recovery for maintenance and reuse', *IEEE Computer*, **7**, 36–49.
Brooks, R. (1983). 'Towards a theory of the comprehension of computer programs', *International Journal of Man–Machine Studies*, **18**, 543–554.
Brown, A. (1993) 'Specifications and reverse engineering', *Journal of Software Maintenance: Research and Practice*, **5**, 147–153.
Burson, S., Kotik, G. and Markosian, L. (1990) 'A program transformation approach to automating software re-engineering', in *Proceedings of COMPSAC '90*, IEEE Computer Society Press, Los Alamitos CA, pp. 314–322.
Byrne, E. J. (1991), 'Software reverse engineering: a case study', *Software—Practice and Experience*, **21**, 1349–1364.
Canfora, G., Cimitile, A. and de Carlini, U. (1992) 'A logic-based approach to reverse engineering tools production', *IEEE Transactions on Software Engineering*, **18**(12), 1053–1064.
Canfora, G., Cimitile, A. and Munro, M. (1994). 'RE²: reverse engineering and reuse re-engineering', *Journal of Software Maintenance: Research and Practice*, **6**(2), 53–72.
Canfora, G., Cimitile, A., Munro, M. and Tortorella, M. (1994) 'A precise method for identifying reusable abstract data types in code', in *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 404–413.
Chen, P. P. (1976) 'The entity-relationship model: towards a unified view of data', *ACM Transactions on Database Systems*, **1**(1), 9–36.
Chikofsky, E. J. and Cross, J. H. (1990) 'Reverse engineering and design recovery: a taxonomy', *IEEE Software*, **7**(1), 13–18.
Darlison, A. G. and Sabanis, N. (1993) 'Data remodeling', in van Zuylen, H. J. (Ed.), *The REDO Compendium: Reverse Engineering for Software Maintenance*, John Wiley & Sons, Chichester, pp. 311–325.
Devanbu, P. (1992) 'GENOA—a customizable, language- and front-end independent code analyser', in *Proceedings of the 14th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 307–319.
Edwards, H. M. and Munro, M. (1995) 'Deriving a logical data model for a system using the RECAST method', in *Proceedings of the 2nd Working Conference on Reverse Engineering. WCRE' 95*, IEEE Computer Society Press, Los Alamitos CA, pp. 126–135.
Estadale, J. and Zuylen, H. J. (1993), 'Views, representations and development methods', in *The REDO Compendium: Reverse Engineering for Software Maintenance*, John Wiley & Sons, Chichester, pp. 93–109.
Gall, H. and Klösch, R. (1995) 'Finding objects in procedural programs: an alternative approach', in *Proceedings of the 2nd Working Conference on Reverse Engineering. WCRE '95*, IEEE Computer Society Press, Los Alamitos CA, pp. 208–216.
Gardarin, G. and Valduriez, P. (1989) *Relational and Knowledge Bases*, Addison-Wesley.
Godart, C. and Charoy, F. (1994) *Databases for Software Engineering*, Prentice-Hall.
Hartman, J. (1992) 'Technical introduction to the first workshop on artificial intelligence and automated program understanding', in *Workshop Notes AAAI-92 AI & Automated Program Understanding*, American Association of Artificial Intelligence, San Jose CA, pp. 8–13.
Jackobson, I. and Lindstrom, F. (1991) 'Re-engineering of old systems to an object-oriented architecture', in *Proceedings of OOPSLA '91*, ACM Press, pp. 340–350.
Jarzabek, S. (1998) 'Design of flexible static program analysers with *PQL*', *IEEE Transactions on Software Engineering*, **24**(3), 197–215.
Jarzabek, S. and Ling, T. W. (1996) 'Model-based support for business re-engineering', *Journal of Information and Software Technology*, **38**(5), 355–374.
Jarzabek, S., Shen, H. and Chan, H. C. (1994) 'A hybrid program knowledge base for static program analysers', in *Proceedings of the Asia–Pacific Software Engineering Conference, APSEC 94*, IEEE Computer Society Press, Los Alamitos CA, pp. 400–409.
Jarzabek, S. and Tan, P. K. (1995) 'Design of a generic reverse engineering assistant tool', in *Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE '95*, IEEE Computer Science, Los Alamitos CA, pp. 61–70.

Kozaczynski, W., Ning, J. and Engberts, A. (1992) 'Program concept recognition and transformation', *IEEE Transactions on Software Engineering*, **18**(12), 1065–1075.

Lano, K. C., Breuer, P. T. and Haughton, H. (1993) 'Reverse engineering COBOL via formal methods', *Journal of Software Maintenance: Research and Practice*, **5**, 13–35.

Linton, M. A. (1984) 'Implementing relational views of programs', in *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, ACM Press, pp. 65–72.

Liu, S. and Wilde, N. (1990) 'Identifying objects in a conventional procedural language: an example of data design recovery', in *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 266–271.

Ohsuga, S. (1995) 'Role of model in intelligent activity—a unified approach to automation of model building', in Kangassalo, H., Jaakkola, H., Ohsuga, S. and Wangler, B. (Eds), *Information Modelling and Knowledge Bases VI*, IOS Press, Amsterdam, pp. 27–42.

Ottenstein, K. and Ottenstein, L. (1984) 'The program dependence graph in a software development environment', in *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, ACM Press, pp. 177–184.

Paul, S. and Prakash, A. (1996) 'A query algebra for program databases', *IEEE Transaction on Software Engineering*, **22**(3), 202–217.

Reps, T. (1994) 'Demand interprocedural program analysis using logic databases', in Ramakrishnan, R. (Ed), *Applications of Logic Databases*, Kluwer Academic Publishers, Boston MA, pp. 163–196.

Rich, C. and Wills, L. (1990) 'Recognising program design: a graph-parsing approach', *IEEE Software*, **1**, 82–89.

Rock-Evans, R. and Hales, K. (1990) *Reverse Engineering: Markets, Methods and Tools*, vol. 1, Ovum Ltd., London.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-oriented Modelling and Design*, Prentice-Hall, New Jersey.

Sneed, H. (1995) 'Reverse engineering as a bridge to CASE', in *Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE '95*, IEEE Computer Society Press, Los Alamitos CA, pp. 300–313.

Sneed, M. H. and Nyary, E. (1995) 'Extracting object-oriented specifications from procedurally oriented programs', in *Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 95*, IEEE Computer Society Press, Los Alamitos CA, pp. 217–226.

Ward, M., Calliss, F. W. and Munro, M. (1989) 'The maintainer's assistant', in *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 307–315.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, **10**(4), 353–357.

Wong, K., Tilley, S., Muller, H. and Storye, M. (1995) 'Structural redocumentation: a case study', *IEEE Software*, 46–54.

Yang, H. and Bennett, K. (1995) 'Acquisition of ERA models from data intensive code', in *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 116–123.

